West Yorkshire Office

GBdirect Ltd
Bradford Design Exchange
34 Peckover Street
BRADFORD
BD1 5BD
West Yorkshire
United Kingdom

consulting@gbdirect.co.uk

Training: 0800 651 0338
General: +44 (0) 870 200 7273
Finance: +44 (0) 1353 615 174

Please call between 0900 and 1700 (UK time) on Monday to Friday

South East Regional Office

GBdirect Ltd
18 Lynn Rd
ELY

8.3. Typedef

Although typedef is thought of as being a storage class, it isn't really. It allows you to introduce synonyms for types which could have been declared some other way. The new name becomes equivalent to the type that you wanted, as this example shows.

```
typedef int aaa, bbb, ccc;
typedef int ar[15], arr[9][6];
typedef char c, *cp, carr[100];

/* now declare some objects */

/* all ints */
aaa     int1;
bbb     int2;
ccc     int3;

ar      yyy;    /* array of 15 ints */
arr     xxx;    /* 9*6 array of int */

c       ch;     /* a char */
cp      pnt;    /* pointer to char */
carr    chry;   /* array of 100 char */
```

The general rule with the use of typedef is to write out a declaration as if you were declaring variables of the types that you want. Where a declaration would have introduced names with particular types, prefixing the whole thing with typedef means that, instead of getting variables declared, you declare new type names instead. Those new type names can then be used as the prefix to the declaration of variables of the new type.

The use of `typedef` isn't a particularly common sight in most programs; it's typically found only in header files and is rarely the province of day-to-day coding.

It is sometimes found in applications requiring very high portability: there, new types will be defined for the basic variables of the program and appropriate `typedef`s used to tailor the program to the target machine. This can lead to code which C programmers from other environments will find difficult to interpret if it's used to excess. The flavour of it is shown below:

```
/* file 'mytype.h' */
typedef short   SMALLINT        /* range *******30000 */
typedef int     BIGINT          /* range ******* 2E9 */
```

Printer-friendly version

The C Book

This book is published as a matter of historical interest. Please read the copyright and disclaimer information.

GBdirect Ltd provides up-to-date training and consultancy in C, Embedded C, C++ and a wide range of other subjects based on open standards if you happen to be interested.

CB6 1DA
Cambridgeshire
United Kingdom
consulting@gbdirect.co.uk
Training: 0800 651 0338
General: +44 (0) 870 200 7273
Finance: +44 (0) 1353 615 174
Please call between 0900 and 1700 (UK time) on Monday to Friday


```
/* program */
#include "mytype.h"

SMALLINT        i;
BIGINT          loop_count;
```

On some machines, the range of an int would not be adequate for a `BIGINT` which would have to be re- `typedef`'d to be `long`.

To re-use a name already declared as a `typedef`, its declaration must include at least one type specifier, which removes any ambiguity:

```
typedef int new_thing;
func(new_thing x){
        float new_thing;
        new_thing = x;
}
```

As a word of warning, `typedef` can only be used to declare the type of return value from a function, not the overall type of the function. The overall type includes information about the function's parameters as well as the type of its return value.

```
/*
* Using typedef, declare 'func' to have type
* 'function taking two int arguments, returning int'
*/
typedef int func(int, int);

/* ERROR */
func func_name{ /*....*/ }

/* Correct. Returns pointer to a type 'func' */
func *func_name(){ /*....*/ }

/*
* Correct if functions could return functions,
* but C can't.
*/
func func_name(){ /*....*/ }
```

If a `typedef` of a particular identifier is in scope, that identifer may not be used as the formal parameter of a function. This is because something like the following declaration causes a problem:

```
typedef int i1_t, i2_t, i3_t, i4_t;

int f(i1_t, i2_t, i3_t, i4_t)/*THIS IS POINT 'X'*/
```

A compiler reading the function declaration reaches point 'X' and still doesn't know whether it is looking at a function declaration, essentially similar to

```
int f(int, int, int, int) /* prototype */
```

or

```
int f(a, b, c, d) /* not a prototype */
```

—the problem is only resolvable (in the worst case) by looking at what follows point 'X'; if it is a semicolon, then that was a declaration, if it is a { then that was a definition. The rule forbidding typedef names to be formal parameters means that a compiler can always tell whether it is processing a declaration or a definition by looking at the first identifier following the function name.

The use of typedef is also valuable when you want to declare things whose declaration syntax is painfully impenetrable, like 'array of ten pointers to array of five integers', which tends to cause panic even amongst the hardy. Hiding it in a typedef means you only have to read it once and can also help to break it up into manageable pieces:

```
typedef int (*a10ptoa5i[10])[5];
/* or */
typedef int a5i[5];
typedef a5i *atenptoa5i[10];
```

Try it out!

Previous section | Chapter contents | Next section